Evolving Dependability

ANDY M. TYRRELL and ANDREW J. GREENSTED
The University of York, UK

Evolvable hardware offers much for the future of complex systems design. Evolutionary techniques not only have the potential for larger solution space coverage, but when implemented on hardware, also allow system designs to adapt to changes in the environment, including failures in system components. This article reviews a number of novel techniques, all based in the field of bio-inspired systems, that provide varying degrees of dependability over and above standard designs. In particular, three different techniques are considered: using FPGAs and ideas from developmental biology to create designs that possess emergent fault-tolerant properties, using FPGAs and continuous evolution to circumvent faults as and when they occur, and, finally, we consider a novel ASIC designed and built with bio-inspired systems in mind.

Categories and Subject Descriptors: B.8.1 [**Performance and Reliability**]: Reliability, Testing and Fault-Tolerance

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Evolutionary algorithms, fault tolerance, bio-inspired architectures, RISA architecture

ACM Reference Format:

Tyrrell, A. M. and Greensted, A. J. 2007. Evolving dependability. ACM J. Emerg. Technol. Comput. Syst. 3, 2, Article 7 (July 2007), 20 pages. DOI = 10.1145/1265949.1265953 http://doi.acm.org/10.1145/1265949.1265953

1. INTRODUCTION

With the increase in system complexity, performing complete fault coverage at the testing phase of the design cycle is very difficult to achieve, if not impossible. In addition, environmental effects such as electromagnetic interference, misuse by users, and the natural ageing of components mean system faults are likely to occur. These faults can cause errors which, if left untreated, could cause system failure. The role of fault tolerance is to deal with the errors caused by

Parts of this work were funded by the EPSRC and the MOD.
A shorter version of this article was presented at the Computing Frontiers Workshop in 2006.
Authors' address: Department of Electronics, University of York, Heslington, YO10 5DD, UK; email: {amt, ajg112}@ohm.york.ac.uk.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2007 ACM 1550-4832/2007/07-ART7 $5.00. DOI 10.1145/1265949.1265953 http://doi.acm.org/10.1145/1265949.1265953

faults in order to avoid failure. Fault tolerance along with fault detection and recovery are techniques used in the design, implementation, and operation of dependable computing systems [Lee and Anderson 1990].

Fault tolerance is increasingly a crucial part of system designs. Many systems have part or all of their function classified as critical in one form or another. Since fully testing a system is generally unrealistic, critical functions must be protected online. This is often achieved by using fault tolerance to cope with errors produced during the operation of the system.

Traditionally, two approaches are taken, both requiring the replication of the system, or system subsections, to be protected. Simple static redundancy (such as N-version systems [Lee and Anderson 1990]) involves the concurrent operation of redundant modules each contributing to a majority decision for a final output. Alternatively, dynamic redundancy operates using a single module, and when a failure is detected or expected, one of the redundant modules is switched into its place. However, these approaches are achieved at the expense of increased equipment needs due to the required replication of hardware and increased design time and costs. These redundancy schemes are termed *space redundancy* as the replicated sections are physically distributed over space. Another category, *time redundancy*, benefits from not requiring a replication of hardware, instead the redundancy is distributed over time. The same operation is repeated, and an output achieved from a consensus of the individual runs. All these redundancy schemes apply equally to a hardware process, a software process, or a combination of both.

Providing continual fault-free operation in a system implies a continual mapping of a logical system onto a nonfaulty physical system. When faults arise, a mechanism must be provided for reconfiguring the physical system such that the logical system can still be represented by the remaining nonfaulty processing elements. Whether the physical platform is a distributed software processor system or consists purely of hard circuitry, for fault tolerance, redundancy in the system's basic processing elements is required. The reconfiguration mechanisms that control utilization of these processing elements can be considered to be based on one of two types of scheme: time-based redundancy reallocation or hardware-based redundancy reallocation.

Time-based use of redundancy involves distributing the function of faulty processing elements among neighboring resources. When reconfiguration occurs, processing elements dedicate some time to performing their own tasks and some to performing the faulty neighbor's functions, possibly resulting in some degradation of the system's performance. In addition, the system operations that are being performed must be sufficiently flexible to ensure their reallocation can be simply performed in real time. Reallocating processes in a hardware redundancy scheme requires spare processing elements and interconnects in order to replace those that become faulty. For this process, reconfiguration algorithms must optimize the use of spares. In the ideal case, a processing system with N spares is able to tolerate N faulty processing elements. However, in practice, this goal is far from being achieved. Reconfiguration of the functional system may not be possible due to limitations of the interconnection capabilities and available resources of each cell.

The majority of hardware redundancy reconfiguration techniques rely on complex algorithms to reassign physical resources to the elements of the logical array. In most cases, these algorithms are executed by a central controller which also performs diagnostic functions and accomplishes the reconfiguration of the physical system. This approach has been demonstrated to be effective, but its centralized nature makes it prone to collapse if the control unit fails. These mechanisms also rely on the designer making a priori decisions on reconfiguration strategies and data/code movement which are prone to error and may in practice be less than ideal. Furthermore, the timing of signals involved in the global control are often prohibitively long and are therefore unsuitable for applying to the control of high-speed systems.

An alternative approach is to distribute the diagnosis and reconfiguration algorithms among all the processing elements in the system. In this way, no central agent is necessary and, consequently, the reliability and time response of the system should improve. However, this decentralised approach has tended to increase the complexity of the reconfiguration algorithm and the amount of communications within the network. In addition, considerable work is required in producing redundancy [Ortega et al. 2000].

Traditionally, fault tolerance has been added explicitly to system designs by including redundant hardware and/or software which take over when an error has been detected. A novel alternative approach would be to design the system in such a way that the redundancy was incorporated implicitly into the hardware and/or software during the design phase. This should provide a more holistic approach to the design process [Ortega et al. 2000; Hollingworth et al. 2000; Canham and Tyrrell 2003; Tyrrell et al. 2001; Bradley and Tyrrell 2002]. We already know that genetic algorithms and genetic programming can adapt and optimize the behavior and structure of solutions to perform a specific task [Fogel 2006], but the aim here is that they should learn to deal with faults within their operation space. This implicit redundancy would make the system response invariant to the occurrence of faults [Thompson et al. 1999; Layzell and Thompson 2000].

This article illustrates a number of novel techniques, all based in the field of bio-inspired electronics, that provide varying degrees of dependability over and above standard designs. In particular, three different techniques are considered: using FPGAs and ideas from developmental biology to create designs that possess emergent fault-tolerant properties, using FPGAs and continuous evolution to circumvent faults as and when they occur, and, finally, we consider a novel ASIC, designed and built with bio-inspired systems in mind, that show how this too can cope with unexpected events during operation.

2. DEVELOPMENTAL TECHNIQUES

Multicellular organisms, the products of long-term biological evolution, demonstrate strong principles for the design of complex systems. Their nascent behaviors, such as growth, cloning (self-replication) and healing (self-repair and fault tolerance), are attracting increasing interest from electronic engineers. All of these characteristics are encoded in the information stored in the genome of the fertilized cell (zygote).

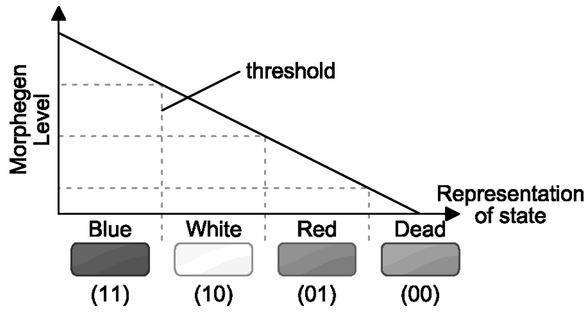

Fig. 1. A cell pattern based on the French flag as originally used by Wolpert to describe cellular development.

The process of growth from a single zygote to a mature organism is called development. Development is controlled by genes which determine the synthesis of proteins. The activity of genes sets up the complex interactions between different proteins, between proteins and genes within cells, and hence the interactions between cells. The development of an embryo is determined by these interactions [Wolpert 2002]. Figure 1 represents a cell pattern whose formation is controlled by the concentrations of a morphogen [Wolpert 2002]. Based on the level of morphogen in the cells, each cell develops into a specific type according to threshold values. These are extremely robust processes and can be subjected to very high levels of failure and still manage to achieve the final goal. This work looks at using development as a way of creating electronic systems based on a cellular array, which we show have similar inherent robustness characteristics [Liu et al. 2005].

2.1 Cell Structure and Intercell Connections

One of the most fundamental features of development is the universal cell structure; each of the cells in a multicellular organism contains the entire genetic material, the genome. Similarly, in the proposed model, each cell is universal and able to perform any function required by the full organism. As shown in Figure 2, every cell only has direct access to the information of its four adjacent cells; no direct long distance interaction between cells is permitted. The internal structure of a cell is shown in Figure 3. Each digital cell is composed of three main components: *control unit* (CU), *execution unit* (EU) and *chemical diffusion module* (CD).

The CU stores information about the cell, including the cell's state (type) and a record of its local virtual chemical level. At every time-step of the model, a next states and chemical generator (NSCG), contained within the CU, determines the cell's next state and next chemical level according to its own and its neighbors current state and chemical level. The operation of generating next state and chemical values is achieved using combinatorial circuits. As shown in Figure 2, these circuits make use of the connections to the cell's 4 immediate neighbors.

The execution unit circuit provides the cell's ability to perform the actual operations of the target application. Using 3-bit-wide signals, data is input

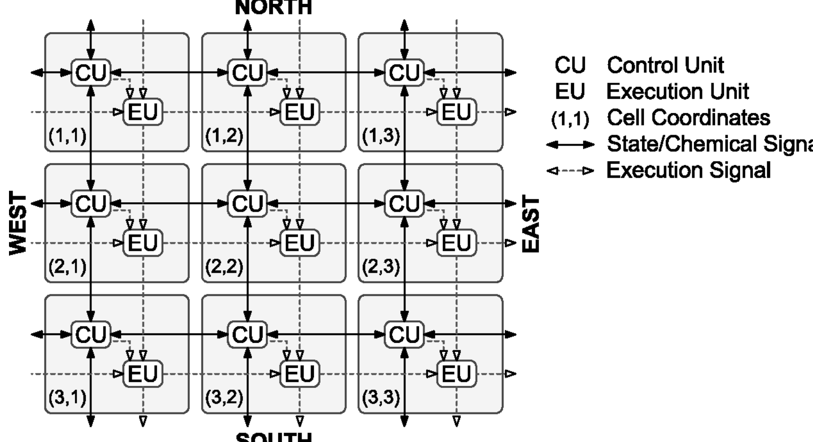

Fig. 2. Interconnection of cells.

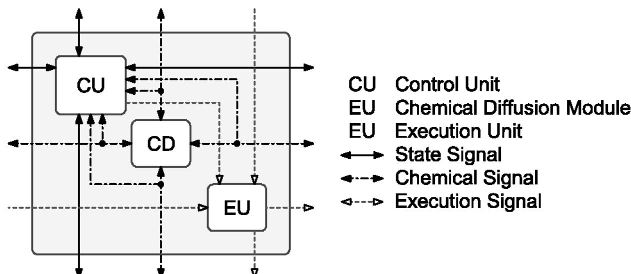

Fig. 3. Digital cell structure.

to each EU from the immediate west and north neighbors and output to the east and south, this configuration avoids the problem of combinatorial feedback paths being created. The state of the cell provides a further input to the EU. The EU uses a 2-bit function selection signal to determine the state, or type, of the cell: 0 denotes a dead cell, other values encode a specific function. Dead cells are those awaiting specialization; they perform no operation and simply propagate their north and west inputs unaltered to their south and east neighbors. At present, only combinational applications are considered, hence the EU is also a combinational circuit.

The internal logical structures of the EU and the NSCG are generated by an evolutionary process, in this case a form of Cartesian genetic programming (CGP) [Miller and Thomson 2000]. The cell's genotype encodes the logic structure of the EU and NSCG.

2.2 Chemical Diffusion

The chemical diffusion module mimics aspects of the real environment in which biological organisms live. As the form of chemical diffusion used in this system would in nature occur within extra-cellular space, a strict biological model would not incorporate this process within the artificial cell. However, it is

more convenient to merge this process into the cell's internal structure. The chemical signal, encoded in 4 bits, enables the transfer of information between cells. The process of chemical diffusion allows cells to send long-distance messages. Furthermore, the chemical signal provides a means to transform a dead cell into an operational living cell. Previous experiments [Liu et al. 2005] suggest that chemicals are indispensable in order to achieve robust solutions; without chemicals; evolved individuals have poor stability and much lower fitness.

The chemical diffusion rule employed in this work is similar to that used in Miller [2003] except in this model only 4 immediate neighbors are used. The rule is given in Equation (1). The chemical level at the next time-step at position (i, j) is calculated from the current chemical level at position (i, j), and those of the 4 immediate neighboring positions (k, l).

$$(C_{ij})_{t+1} = \frac{1}{2}(C_{ij})_t + \frac{1}{8}\sum_{k,l \in N}(C_{kl})_t. \tag{1}$$

The result is that each cell retains half of its previous chemical and distributes the other half equally to its four adjacent cells and receives the diffused chemical from them. Governed by this diffusion equation, it is the task of the chemical diffusion module to perform diffusion by updating a cell's chemical level and propagating the new calculated value to the four adjacent cells.

2.3 Digital Organism Growth

Starting with an evolved genotype that specifies the cellular structure, a zygote is placed at the center of the grid-based artificial environment. Initially, apart from the zygote cell, all other cells are dead (in state 0). The central positioning of the zygote speeds up the growth as it takes the least time for the digital

Algorithm 1. Organism Growth Algorithm

```
 1: Initialise zygote and environment chemical.
 2: repeat
 3:    Perform chemical diffusion.
 4:    Simultaneously update the cells' state.
 5:    if no chemical exists at a position or the cell and
         its four neighbours are dead then
 6:      Cell's genotype encoded program is not executed
 7:    else
 8:      Cell's genotype encoded program is executed
         generating the cell's next chemical and state values.
 9:    end if
10:    if the next state generated is alive then
11:       Overwrite this position's chemical with new chemical value.
12:    else
13:       Do not alter the chemical at this position.
14:    end if
15: until the stopping criterion has been reached.
```

Table I. Available Molecule Functions

Name	Algebraic Expression	Circuit
MUX1(A,B,C)	$AC + B\overline{C}$	A, B, C, 0, 1
MUX2(A,B,C)	$\overline{A}C + B\overline{C}$	A, B, C, 0, 1
MUX3(A,B,C)	$AC + \overline{B}\,\overline{C}$	A, B, C, 0, 1
MUX4(A,B,C)	$\overline{A}C + \overline{B}\,\overline{C}$	A, B, C, 0, 1

organism to cover the whole environment. The inputs to the cells located at the border of the environment are fixed to 0. The cells in the model require the presence of chemicals to live, therefore, some initial chemicals must be injected at the position of the zygote. Given a genotype, the growth procedure is described in Algorithm 1.

The model used in this article was inspired by software simulation of the French Flag problem [Wolpert 2002]. However, this experiment incorporates the EUs to enable a more practical application. The first such application, chosen due to its simplicity, was a 2-bit multiplier. The task was to evolve a cell circuit that would grow to become a 3×3 cell organism implementing a 2-bit multiplier. The inputs to the multiplier were connected to the execution signals of cell (1, 1) and (2, 1), while the output execution signals of cell (2, 3) and (3, 3) drove the output result.

2.4 Evolving Molecules

The molecule is the fundamental element of the cell's evolved components. Each evolvable subcircuit (the EU and NSCG) are composed of several molecules, each representing a different logic gate. The cell's genotype encodes the choice and connectivity of these molecules/gates forming a complete circuit.

The molecule used in this experiment is a 3-input universal logic module (ULM). The ULM is formed by a 2-1 multiplexer. This choice allows any 2-input logic function to be implemented. Higher order ULMs could have been chosen, however, larger fan-in cells incur an increase in wiring density and complexity [Miller 2003].

The MUX circuit, described in Equation (2) when combined with negated input variables and constant input values, is able to realize all 2-input functions. However, the available molecule functions used in this experiment are limited to the four shown in Table I.

$$f(A, B, C) = AC + B\overline{C}. \tag{2}$$

The available inputs to a molecule in the hardware implementation are constrained. As a result, constant logic values are not available as molecule inputs. However, a constant value can be generated by an upstream MUX. MUX2 with

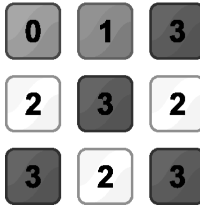

Fig. 4. Evolved cell state pattern for 2-input multiplier.

the same value applied to all inputs will generate a logic 0, whereas a similarly connected MUX3 generates a logic 1.

2.5 Evolution Strategy

Two evolutionary phases are used to create the complete organism genotype.

(1) The first phase evolves the EU circuit and the 3×3 cellular structure required to implement that circuit. The genotype in this phase consists of two parts: the CGP part encodes the EU structure, while the other is the states of the organisms 9 cells. The fitness is measured by the number of correct bits of the multiplier result output. The multiplier has two 2-bit inputs, so it has a total of $2^4 = 16$ possible outputs values. As the multiplier has a 4-bit output, there are $16 \times 4 = 64$ bits with which to judge a correct multiplier truth table. Therefore, the maximum fitness value is 64.

(2) The second phase evolves the structures of the NSCG circuits such that a stable organism will develop into the cellular structure evolved in Phase (1). This phase is the same as the evolution process described in the French Flag problem, except for some parameter values [Miller 2003].

One of the patterns found in the first phase is shown in Figure 4. This pattern utilizes all available cell positions with a diverse and complete distribution of states. It was chosen as the target configuration of the digital organism along with its corresponding EU structure obtained via evolution.

2.6 Hardware Implementation and Fault Injection

A FPGA-based implementation of the organism was tested using Xilinx® hardware and tool-chain. The organism's developmental process is shown in Figure 5[1]. It can be seen that the organism matures at 1ns when the state pattern is identical to that shown in Figure 4.

To test the fault-tolerance abilities of the evolved organism, transient faults were injected into the organism. After the organism had stabilised to the correct structure shown in Figure 4, the chemical level at cell positions (2, 1), (2, 2), (2, 3), and (3, 3) were simultaneously reset to 0. Figure 6 shows the organism's recovery. As the organism continues its growth, it recovered flawlessly at 2.4ns, resulting in the multiplier output regaining the correct value.

[1]To simplify the creation of these plots, a ModelSim simulation was used to illustrate the hardware operation.

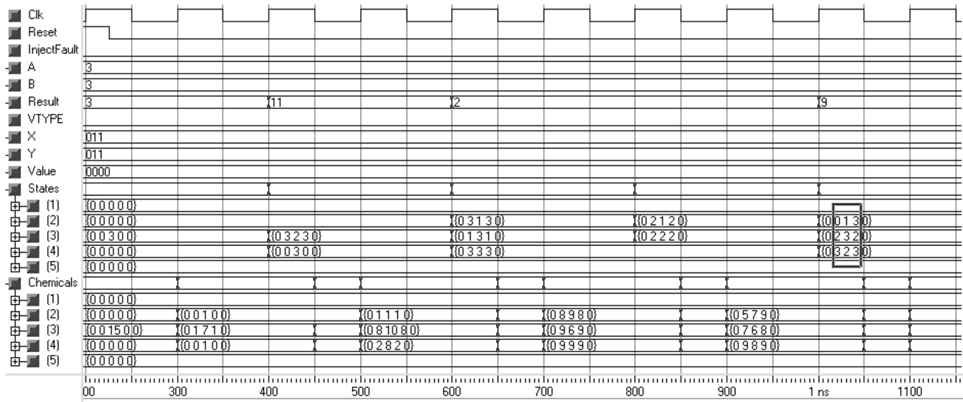

Fig. 5. Developmental growth procedure. Traces A and B show the multiplier inputs, Result shows the output. The evolution of the cell state pattern can be seen in traces States (2:4). The pattern for a working multiplier is boxed and is also shown in Figure 4. The changing chemical levels are also shown.

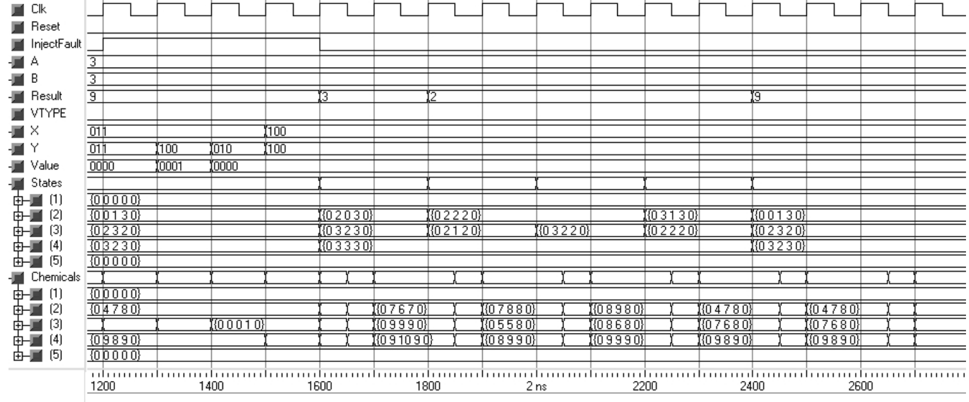

Fig. 6. Injection of the first set of faults (chemical disruption) and the recovery procedure.

Once the organism had again stabilized to the correct structure, the state of the cells at positions (2, 1), (2, 3), and (3, 1) were simultaneously reset to 0 (dead cell). Figure 7 shows the recovery process after this second set of transient faults. The states of the 3 selected victim cells recover completely to the correct pattern at 4ns.

3. CONTINUOUS EVOLUTION

The majority of evolutionary algorithms stop when an acceptable solution has been found. However, in reality, evolution is a continuous process. Here we show that by using this continuous evolutionary approach, it is possible to produce systems that will continue to operate successfully even in the presence of faults [Tyrrell et al. 2004]. This principle is demonstrated using a simple robotic application, although the results should be generic to any

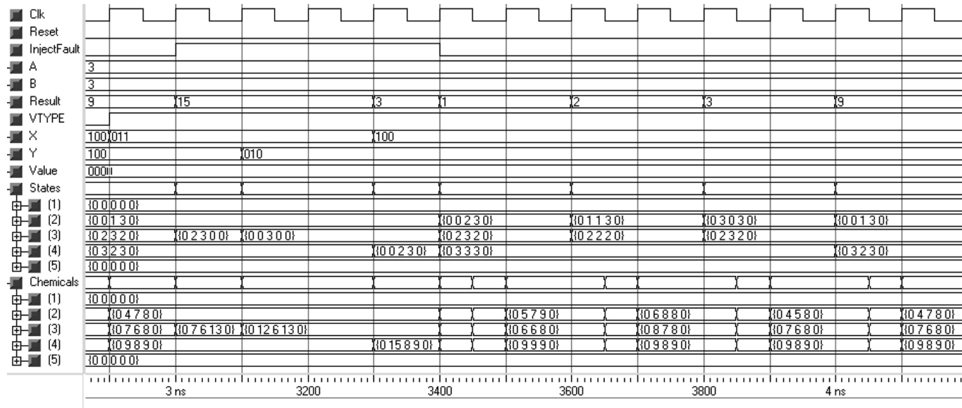

Fig. 7. Injection of the second set of faults (state disruption) and the recovery procedure.

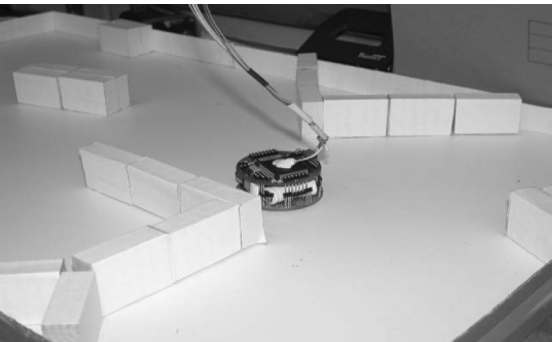

Fig. 8. The Khepera robot performing obstacle avoidance.

application. The experiments, detailed in the following, use a Khepera robot [KTeam] as a platform for the continuous evolution of an object avoidance controller (Figure 8).

The novel evolutionary algorithm used in this work is made up of a population of parents and a number of subpopulations created by cloning parents. These clone populations are used to search for innovative solutions that track dynamic changes in the environment. It is plausible from an engineering view point that the innovation populations should undergo mutation at a rate derived from their fitness, thus reflecting the adaptability of the individual to that environment. Mutation is therefore carried out as follows: individuals with high fitness are subject to a low mutation rate, while individuals with low fitness are subjected to a high mutation rate. For the robot navigation problem, the mutation rate, defined according to an individual's fitness, is given in Equation (3)

$$n = k(1 - norm_fit), \tag{3}$$

where n is the number of bits to be mutated, k is a constant to be tuned by the user, and *norm_fit* is the normalized fitness for the population of parents.

Evolution starts with an initially randomized parent population P. This population is cloned μ times, resulting in μ cloned populations, C, each the same size as the original parent population. The cloned populations undergo mutation, resulting in μ mutated populations, C'. The mutated clone populations and parent population are evaluated and then ordered in a single list according to fitness. The fittest individuals from the list are used to create the parent population for the next generation, the remaining individuals are discarded. The parent population, P, is not mutated in order to make sure that good parent individuals are not lost by the mutation operation this way, a form of elitism is undertaken.

Choosing an appropriate fitness measure is an important part of designing an evolutionary algorithm. Special consideration must be taken for the robot controller fitness measure as individuals will encounter different aspects of a dynamic environment. For our robot navigation problem, the controller's fitness is determined by simply measuring the time and distance the robot has run before hitting an obstacle; the more time elapsed without collision and the greater the distance traveled, the higher the fitness value. This fitness measure is calculated as in Equation (4)

$$fitness = distance \times \frac{time}{1000}. \quad (4)$$

Distance is measured using wheel rotation counters built into the robot. A distance of 1000 represents a movement of approximately 40mm. If the robot is turning, no adjustment to the distance value is made. To limit the time that each controller is trialed, a maximum time limit of 140s is imposed. Individuals are killed when this limit is reached even if a collision has not occurred. Furthermore, if an individual is stuck in the same position without any distance improvement, it is killed. The fitness limit in the experiment is about 1400, which means the robot moved forward without any steering change. At the start of evaluating a new individual, an escape time is provided to allow movement away from a potential dead zone where the last individual was killed. Clearly, since only one robot is used, only one individual is running at a time.

3.1 Experimental Results

For the following experiments, a parent population size of 16 was used; this was cloned three times ($\mu = 3$) to result in a total population size of 64 ($16+(3\times16)$). In order to compare the results, a baseline experiment was performed where the individuals of the cloned populations experience a constant mutation rate of 8 bits per-bit string. Figure 9 shows the results for this experiment. All results presented are for the population of parents, and the fitness values are average values of 10 runs.

The next experiment makes use of the mutation rate defined by Equation (3). It can be observed from the results in Figure 10, where $k = 16$, that the mutation operator is very important to evolve a controller for autonomous robot navigation. It might be argued that it is quite difficult to obtain adaptive behavior with a constant mutation rate.

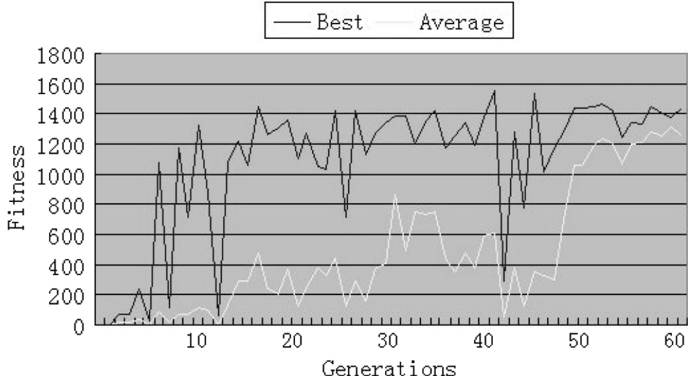

Fig. 9. Behavior for the best and average fitness of the population of parents (P) when a constant mutation rate was applied to the individuals of the cloned populations (C).

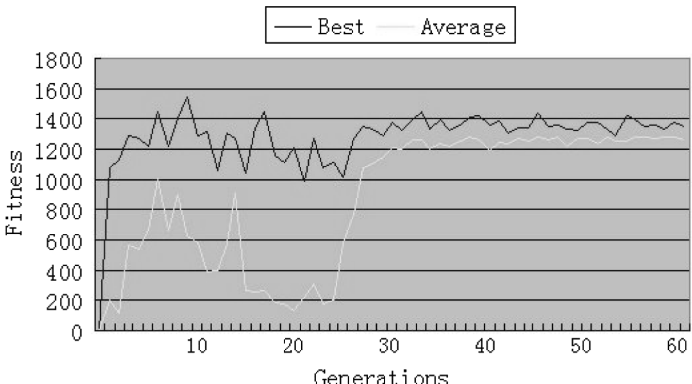

Fig. 10. Behavior for the best and average fitness of the population of parents, P, when the mutation rate for the individuals of the cloned populations (C) was calculated according to 16*(1-norm_fit).

3.2 Fault Injection

The following results demonstrate the effects of applying faults to the robot's proximity sensors. This can be considered equivalent to the dynamic nature of the environment affecting sensory data due to noise or sensor failure. For the following experiments, the maximum number of bits mutated in the cloned populations is set to $k = 16$. Faults were introduced by covering the robot's proximity sensors used for navigation with paper masks. This results in the blocked sensor outputting a fixed reading. The results shown in Figure 11 illustrate the effect of the robot having a single failed sensor throughout the experiment. This shows that the algorithm is still able to evolve an effective controller with a reduced number of sensory inputs.

For the next experiments a fault was added at runtime. Once an acceptable controller had been evolved, a fault was applied to one of the robot's proximity sensors. Figure 12 shows the result of applying a sensor fault at generation 50. This fault disrupted the whole population of controllers. It can be observed that

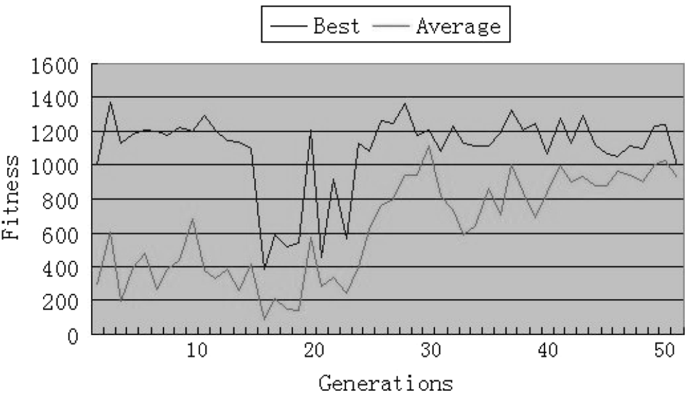

Fig. 11. Behaviour for the best and average fitness of the population of parents, P, when a proximity sensor fault is applied before evolution starts.

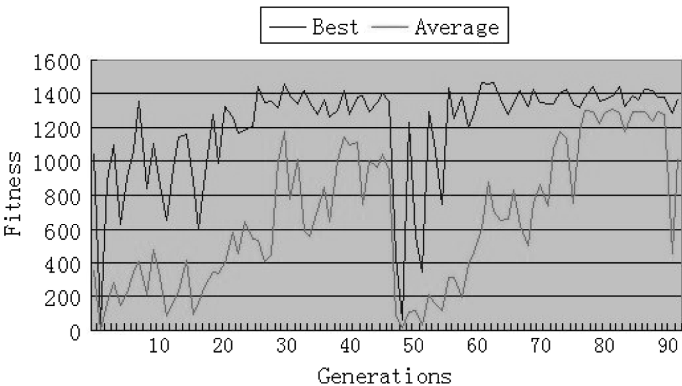

Fig. 12. Behavior for the best and average fitness of the population of parents, P, when a proximity sensor fault is applied at generation 50.

both the average fitness and the fitness of the best individual dropped when the fault was applied. However, after around 10 generations, the fitness of the best individual climbs up and regains its original value. This result demonstrates that while immediate failure is not prevented, when a fault occurs, the system does quickly recover as fitness is regained.

4. DEDICATED HARDWARE

The previous two sections have discussed techniques for increasing dependability using evolutionary and developmental methods implemented on commercial FPGAs. In this final example, dependable architectures using hardware that has been specifically designed to assist evolutionary methods will be considered.

The goal of the project was to design and build an ASIC with features amenable for bio-inspired work and, in particular, embryonics and dependable system design. The main novel features, setting the architecture and device apart from conventional programmable devices, are the following.

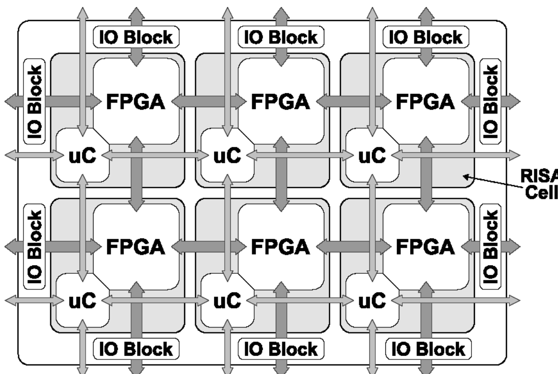

Fig. 13. The RISA architecture. A two-dimensional array of highly reconfigurable RISA cells.

—A distributed routing algorithm that performs automated creation of connections between parts of the chip.
—The ability to perform intrinsic partial reconfiguration without disrupting circuit operation.
—A random configuration-safe architecture such that random configuration bitstreams will not damage the chip.
—A microcontroller array with a dedicated microcontroller per-device cell.

The Reconfigurable Integrated System Array (RISA) [Greensted and Tyrrell 2007] is a novel form of field programmable digital device. The architecture is formed by a two-dimensional array of RISA Cells, each containing a microcontroller and a section of FPGA; this combination forms the Integrated System referred to by the architecture name. An illustration of the RISA architecture is shown in Figure 13.

The RISA architecture is inspired by the structure of biological organisms. In particular, the microcontroller/FPGA combination used within each RISA cell reflects the structure and operation of real cells. As biological cells perform system functions and control the embryonic development process, the electronic RISA cell must also contain this functionality. The RISA cell's microcontroller acts as the cell nucleus. It is able to control cell specialization using stored configurations, analogous to DNA, to implement different cell functions, depending on cell location. The FPGA fabric then provides the platform on which subsections of the system circuitry may be implemented. The mapping between biological and electronic domains is illustrated in Figure 14.

Past embryonic architectures have been implemented on FPGAs [Ortega et al. 2000]. However, despite their role as quick prototyping and testing platforms, the inefficiency of implementing a reconfigurable platform within another means commercial FPGAs are not suitable for the RISA architecture. Therefore, an ASIC implementation of the RISA design has been fabricated. Two particular design features necessitating a custom ASIC are the microcontroller array and an FPGA fabric that supports fine-grained partial

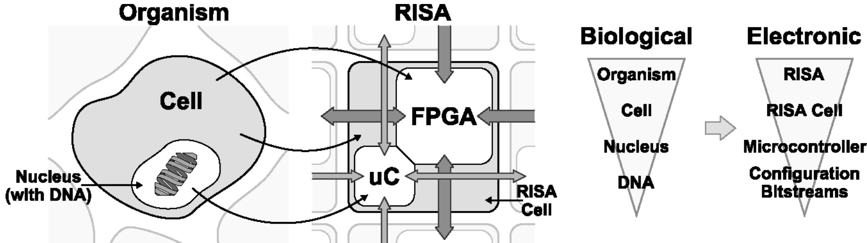

Fig. 14. The biological to electronic mapping of the RISA architecture. The RISA cell must contain sufficient functionality to perform the cell's function and to drive differentiation.

reconfiguration. These two main aspects of the RISA design are detailed in Sections 4.1 and 4.2, respectively.

4.1 SNAP Microcontroller

Each RISA cell microcontroller is responsible for the functional specialization of the cell. This involves determining the cell's position within the system and configuring the cell's FPGA fabric accordingly. Rather than using a hard-coded state machine to control this process, a microcontroller core has been employed to provide the flexibility for investigating different differentiation methods. Furthermore, this approach allows other processor array-based applications to be implemented on the RISA architecture.

A custom microcontroller core was developed for the RISA architecture. The Simple Networked Application Processor (SNAP) is a 16-bit RISC core with a number of peripheral modules providing extra functionality. The SNAP core is illustrated in Figure 15. As already mentioned, the main role of the microcontroller is to control the configuration of the cell's FPGA fabric. In order to simplify this task, the microcontroller contains dedicated configuration access ports (CAPs) to access the FPGA configuration chains. Furthermore, the SNAP instruction set has been specifically designed to include instructions that simplify the task of bitstream creation and manipulation. A set of peripheral modules are available to the user via the core's register file. Modules include a random number generator, general purpose counters, and a UART. To simplify the task of creating a SNAP network, each core includes four SNAPLink modules. These provide independent, full duplex, flow controlled connectivity to all four neighboring cells. Intermicrocontroller communication is as simple as reading from and writing to the SNAPLink registers.

The SNAP microcontroller uses a von Neumann memory architecture. This approach makes more efficient use of the core's limited memory by avoiding wasted program memory space that could be used for data storage. To further reduce memory requirements, the SNAP design promotes compact assembler code. This is achieved in two ways. First, all SNAP instructions can be made conditional on the state of core status flags. This approach helps remove the need for separate test instructions prior to a branch. Second, by accessing all peripheral modules through the register file, peripheral data may be directly

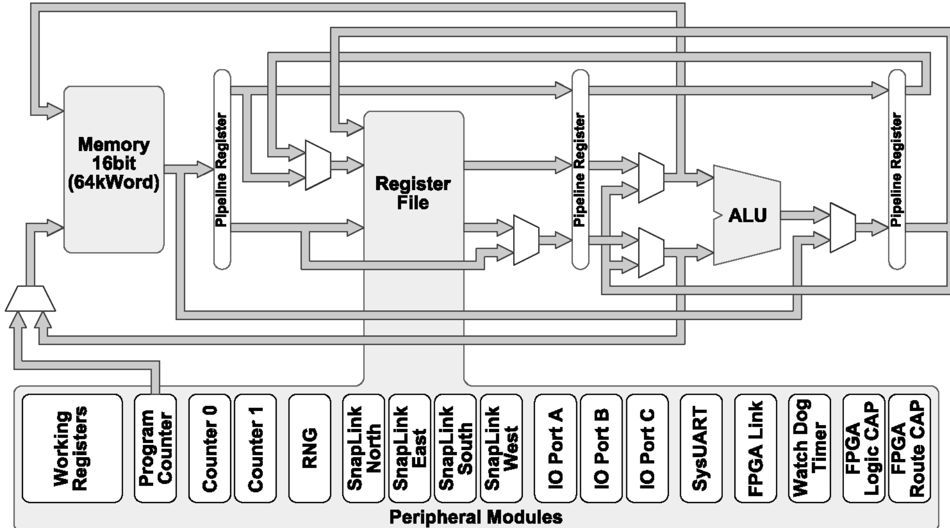

Fig. 15. The SNAP microcontroller, in summary, a 16-bit data width, 16-bit address space, 4-stage pipelined, RISC, von Neumann memory design. Peripheral modules, accessed via the register file, provide extra functionality.

written to or read from by all instructions, without having to also use memory read and write instructions.

4.2 FPGA Fabric

The FPGA fabric used with the RISA architecture, like any commercial FPGA, should be sufficiently flexible to implement a multitude of different circuit designs. However, unlike standard FPGA devices, the RISA architecture uses a considerably more flexible configuration system. This satisfies the requirement of embryonic arrays for fine-grained partial reconfiguration, the ability to target and reconfigure small areas of the FPGA fabric without having to take the remaining areas offline.

The RISA configuration scheme attempts to provide flexibility and simplicity to the designer. Reconfiguration takes place without affecting the current configuration. Furthermore, once the new configuration data is in place, it can be made active, replacing the old configuration in a single clock cycle. FPGA logic configuration is separated from that of the routing, which makes reconfiguration faster if only the logic or routing needs altering. It is also possible, using the configuration circuitry, to read back a snapshot of FPGA register values.

An important design feature of the RISA FPGA architecture is the ability to safely perform random configuration manipulation without risking device damage. It is the routing architecture that makes this possible. RISA routing is multiplexer-based rather than bus-based. This approach removes the possibility of bus contentions that may lead to device damage. Furthermore, the routing design removes the possibility of creating combinatorial feedback paths. This property makes the RISA architecture ideal for investigating evolvable hardware designs.

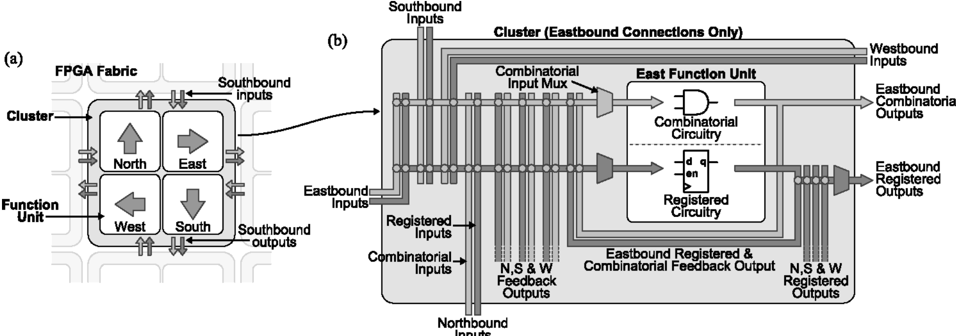

Fig. 16. The FPGA fabric is formed by an array of clusters. Each cluster contains four function units that contain the configurable logic elements.

To achieve a random configuration-safe architecture, the FPGA fabric is organized into four combinatorial directions. The main configurable element of the FPGA is the cluster. Each cluster contains four function units assigned to a different direction, north, east, south and west. Intercluster combinatorial connections can only be made between function units of the same direction. In this way, it is not possible to form a combinatorial loop. Direction changes must be made via a registered connection. The FPGA fabric is illustrated in Figure 16.

4.3 The RISA Chip

The RISA architecture has been fabricated using UMC's 0.18μm 1P6M logic process. A cell-based approach was used as it provides a simple method for a first ASIC run [Smith 1997]. Due to the density limits this approach imposes, the first RISA chip only implements a single RISA Cell. The greater portion of the chip's die area was used in implementing the FPGA, the density of which could be substantially improved by taking a full custom approach especially that of the FPGA routing circuitry. However, as the RISA architecture is array-based, it has been specifically designed to allow the abutment of separate devices to create larger arrays. This means that the current chip is still appropriate for forming a full RISA architecture with multiple RISA cells.

4.4 Specialization and Routing Algorithms

Traditional embryonics uses a simple Cartesian grid to determine cell location and thus specialization [Jackson 2003; Mange et al. 2000]. However, as this approach requires the removal of a complete array row or column in order to replace a faulty cell, repair is wasteful of resources. The approach taken in this work uses a more efficient routing algorithm based on ideas from computational topology and geometry. The algorithm performs initial routing and subsequent rerouting for fault recovery. The algorithm makes use of discrete Morse-Lyapunov functions to track the availability of chip resources, that is, the utilization of RISA cells. The algorithm operates on rooted binary trees bound within a finite region of a Euclidean plane. In a purely distributed

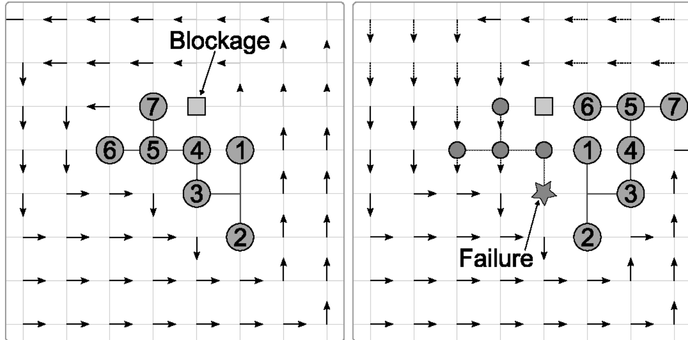

Fig. 17. The cell routing algorithm in operation. The images shows how a graph of connected cells is relocated and rerouting after a cell failure has occurred. The blockage shows how the graph has to be placed around dead cell locations.

fashion, the algorithm assigns weighting values to each node in the graph. These values represent the size of a particular node's subtree. When reconfiguration is required, the weightings are used by each graph node to determine a path for graph redirection. In this way, the algorithm provides the ability to perform complex deformation of the original circuit structure in a distributed manner while maintaining the system's functional integrity. Figure 17 shows an example of a graph that has rerouted after a node has been failed.

5. CONCLUSION

This article has described three different directions in evolvable hardware research; an evolved development circuit to drive the self-organization of a multicellular system, an algorithm for continuous evolution in order to achieve continual adaptation to a changing environment, and a new hardware platform specialized for investigating evolutionary processes.

The evolution of a developmental system has been described where a digital system is grown via embryonic development of digital hardware cells. Morphogenic chemicals are used to mimic the pattern formation process of embryonic growth. Artificial evolution was used to develop the internal circuitry of cell execution units that provide cell function and next state and chemical generators that drive the differentiation of the organism. Experimental results have demonstrated that the developed system is able to tolerate transient faults in the form of cell state and chemical-level disruptions. After the injection of these faults, the circuit is able to regrow back into a fully functional circuit.

The continual evolution research reflects that in nature the evolution of a species does not stop when strong individuals are bred, rather evolution continues to allow subsequent generations to adapt to environmental changes. Experimental results using a Khepera robot have shown that a continual evolution algorithm is suitable for creating an autonomous navigation controller for collision avoidance even with the occurrence of sensor faults. The following important features with regard to autonomy, adaptability, and robustness were observed.

—By using a mutation rate defined according to the fitness of the individual, it is possible to obtain good adaptability in changing environments.
—A robust controller undergoing continual evolution can reevolve in order to continue operation in the presence of faults.

It was demonstrated that despite the introduction of permanent faults, the evolutionary process ensures the system recovers to full functionality. Such an ability would be of great benefit to applications such as unmanned space and underwater missions where the lack of an adaptation mechanism could lead to mission failure in the event of system faults.

Finally, a description of the RISA architecture, a new FPGA-like device designed specifically for use within the bio-inspired community, has been given. The device provides an advanced reconfigurable hardware platform suitable for investigating embryonics. Furthermore, the ability to safely use random configuration bitstreams and be able to target specific areas of the device's reconfigurable fabric makes the RISA chip far more suitable than commercial devices for evolutionary hardware research.

A routing algorithm designed for use with the RISA architecture has been described. Compared to traditional embryonic architecture, this algorithm makes more efficient use of system resources when reconfiguring a system to circumvent faults.

ACKNOWLEDGMENTS

The authors would like to thank Julian Miller, Heng Liu, Natasha Dowding and Renato Krohling for their help and advice.

REFERENCES

BRADLEY, D. AND TYRRELL, A. 2002. Immunotronics: Novel finite state machine architectures with built in self test using self-nonself differentiation. *IEEE Trans. Evolut. Computat. 6*, 3, 227–238.

CANHAM, R. AND TYRRELL, A. 2003. An embryonic array with improved efficiency and fault tolerance. In *5th NASA Conference on Evolvable Hardware*. IEEE Computer Society, 265–272.

FOGEL, D. 2006. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. Wiley-IEEE Press, Piscataway, NJ.

GREENSTED, A. AND TYRRELL, A. 2007. RISA: A hardware platform for evolutionary design. In *Proceedings of IEEE Workshop on Evolvable and Adaptive Hardware*. IEEE Computer Society, 1–7.

HOLLINGWORTH, G., SMITH, S., AND TYRRELL, A. 2000. Safe intrinsic evolution of virtex devices. In *Proceedings of the 2nd NASA/DoD Workshop on Evolvable Hardware*, J. Lohn, A. Stoica, and D. Keymeulen, Eds. IEEE Computer Society, 195–202.

JACKSON, A. 2003. Asynchronous embryonics, self-timed biologically-inspired fault-tolerant computing arrays. Ph.D. thesis, The University of York.

KTEAM. Khepera Web site. `http://www.k-team.com`.

LAYZELL, P. AND THOMPSON, A. 2000. Understanding inherent qualities of evolved circuits: Evolutionary history as a predictor of fault tolerance. In *Proceedings of the 3rd International Conference on Evolvable Systems (ICES'00)*. Lecture Notes in Computer Science, vol. 1801. Springer-Verlag, Berlin, Germany, 133–142.

LEE, P. AND ANDERSON, T. 1990. *Fault Tolerance Principles and Practice* 2nd ed. Springer-Verlag, Berlin, Germany.

Liu, H., Miller, J., and Tyrrell, A. 2005. Intrinsic evolvable hardware implementation of a robust biological development model for digital systems. In *6th NASA Conference on Evolvable Hardware*. IEEE Computer Society, 87–92.

Mange, D., Sipper, M., Stauffer, A., and Tempesti, G. 2000. Towards robust integrated circuits: The embryonics approach. *Proceedings of the IEEE 88*, 4 (April), 516–541.

Miller, J. 2003. Evolving developmental programs for adaptation, morphogenesis, and self-repair. In *7th European Conference on Artificial Life*. Lecture Notes on Artificial Life, vol. 2801. Springer-Verlag, Berlin, Germany, 56–265.

Miller, J. and Thomson, P. 2000. Cartesian genetic programming. In *Genetic Programming, Proceedings of EuroGP'2000*, R. Poli, W. Banzhaf, W. Langdon, J. Miller, P. Nordin, and T. Fogarty, Eds. Lecture Notes in Computer Science, vol. 1802. Springer-Verlag, Berlin, Germany, 121–132.

Ortega, C., Mange, D., Smith, S., and Tyrrell, A. 2000. Embryonics: A bio-inspired cellular architecture with fault-tolerant properties. *Genetic Program. Evolv. Machines 1,* 3, 187–215.

Smith, M. 1997. *Application-Specific Integrated Circuits*. Addison Wesley Professional Publishing, Boston, MA.

Thompson, A., Layzell, P., and Zebulum, R. S. 1999. Explorations in design space: Unconventional electronics design through artificial evolution. *IEEE Trans. Evolution. Computat. 3,* 3, 167–196.

Tyrrell, A., Hollingworth, G., and Smith, S. 2001. Evolutionary strategies and intrinsic fault tolerance. In *Proceedings of the 3rd NASA / DoD Workshop on Evolvable Hardware*. IEEE Computer Society, 98–106.

Tyrrell, A., Krohling, R., and Zhou, Y. 2004. A new evolutionary algorithm for the promotion of evolvable hardware. *IEE Proceedings of Computers and Digital Techniques 151*, 4 (July), 267–275.

Wolpert, L. 2002. *Principles of Development* 2nd ed. Oxford University Press, Oxford, UK.

Received October 2006; revised January 2007; accepted February 2007 by Sally McKee